

Enterprise Application Development – Lab 3

C15478448



Enterprise Application Development

Course: DT228/4

Lab 3



Table of Contents

**Part 1 – GRAPHQL Schema 3**

**Part 2 – GraphQL Query Resolver 12**

**Part 3 – Three Joined Database Relations 16**

**Part 4 – Mutation Resolver 18**

**Part 5 – GraphQL Server and TESTS 19**

Folder Structure

Prisma-GraphQL – Contains All Parts (1-5)

Tutorial – Contains the tutorial for Prisma GraphQL

Document – This Document in .docx and .pdf

Screenshots – Screenshots for all outputs

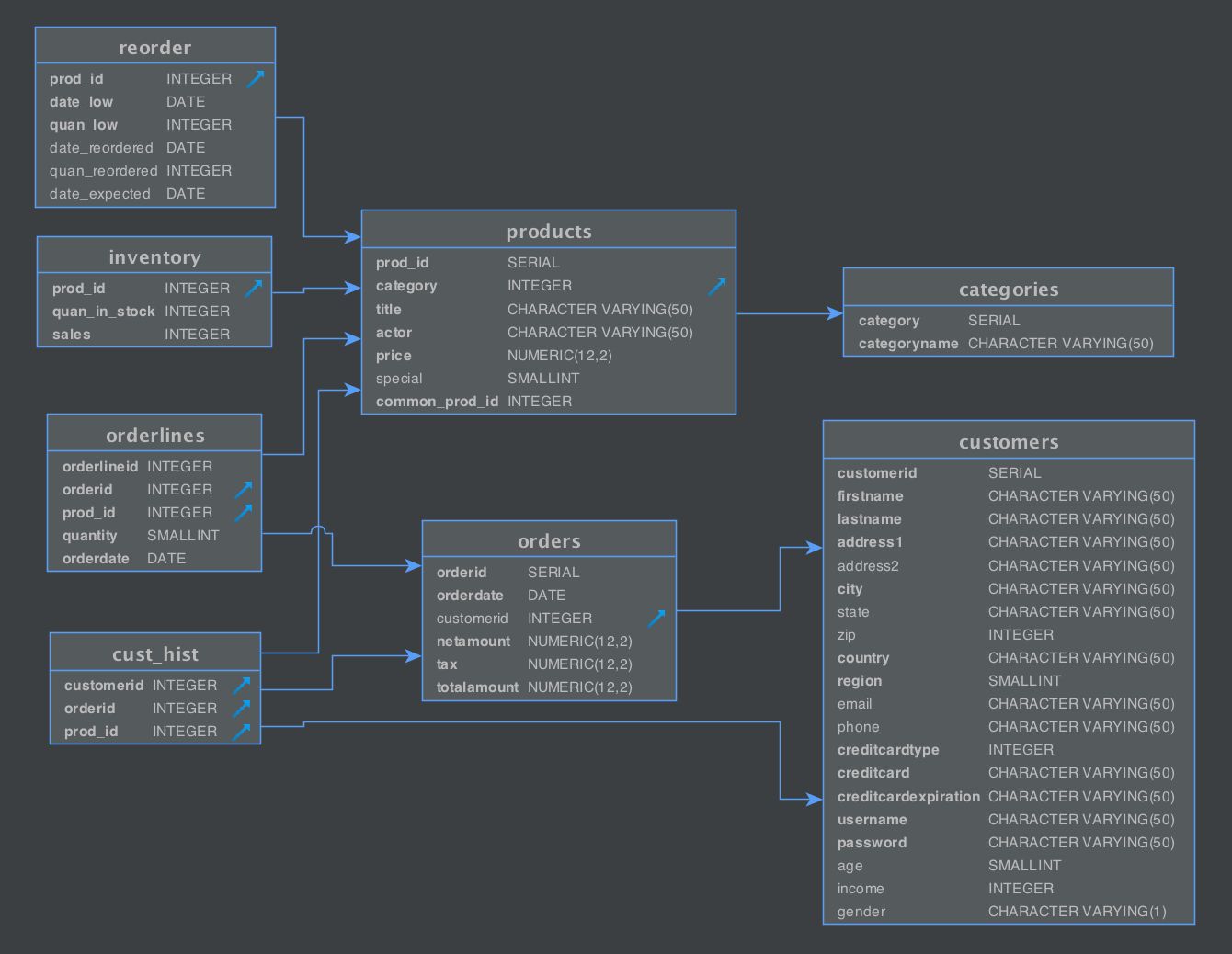
**Note:**

* To run this project you must enter in the following command in the **Prisma-GraphQL** folder: *npm install*
* After installing the required modules based on the package, run the following command: *node index.js*
* Enter: *localhost:4000* into a browser for relevant links to showcase each part.

**Problem Sets**

**Part 1 – GraphQL Schema**

*Using graphql-yoga and the ERD below, construct a graphql schema using any four relations of your choice having the relationships depicted.*



Upon starting, a new folder must be created which in my case I made a folder and named it **Prisma-GraphQL**, upon doing so I then created a **Docker Compose** file, this is to launch Prisma on my machine, it’s important to have this file as it configures Prisma and specifies the database it can connect to.

Inside the file named **docker-compose.yml** I added relevant code to add Prisma and the database docker images. I picked **PostgreSQL** as my database.

I launched Prisma and the connected database with the following command**: docker-compose up -d**.

I then configured my Prisma API as I needed to bootstrap the configurations files for my Prisma client, I did so with the following command: **prisma init --endpoint** [**http://localhost:4466**](http://localhost:4466).

The **prisma init** command created my minimal setup that I need to deploy my Prisma datamodel with the following files: **prisma.yml** and **datamodel.prisma**.

The file named **datamodel.prisma** is where the schema for the above image from the question will be created in. I created all eight tables with the appropriate relations for each table.

# Customers Table

type Customers {

id: ID! @unique

firstname: String!

lastname: String!

address1: String!

address2: String

city: String!

state: String!

zip: String

country: String!

region: String

email: String! @unique

phone: String

creditcardtype: String

creditcard: String

creditcardexpiration: String

username: String! @unique

password: String!

age: Int!

income: Float

gender: String!

}

# Orders Table

type Orders {

id: ID! @unique

orderdate: DateTime!

netamount: Float!

tax: Float!

totalamount: Float!

customers: Customers

}

# Categories Table

type Categories {

id: ID! @unique

categoryname: String!

}

# Products Table

type Products {

id: ID! @unique

title: String

actor: String

price: Float

special: Boolean

common\_prod\_id: Int

categories: Categories

inventory: Inventory

}

# Re-order Table

type Reorder {

id: ID! @unique

date\_low: DateTime!

quan\_low: Int

date\_reordered: DateTime!

quan\_reordered: Int

date\_expected: String

products: Products

}

# Inventory Table

type Inventory {

id: ID! @unique

quan\_in\_stock: Int

sales: Int

products: Products

}

# Order Lines Table

type Orderlines {

id: ID! @unique

quantity: Int

orderdate: DateTime!

orders: Orders

products: Products

}

# Customer History Table

type Cust\_hist {

id: ID! @unique

customers: Customers

orders: Orders

products: Products

}

Then, I ran the following command: **prisma deploy** to finish the setup for my database schema, this is needed for the Prisma client to talk to my database from code.

Whenever the **datamodel.prisma** file is changed (aside from adding comments) you must always deploy it as it will need to generate updates.

Now that the datamodel is complete, I needed to deifine a GraphQL schema for my GraphQL server in other words I had to define my GraphQL API.

In the **schema.graphql** file I specified the **Query** type in my GraphQL Schema.

###

# QUERY

###

# Querying the Database

type Query {

# Customers

allCustomers: [Customers!]! # Retrieve all Customers

specificCustomers(customersId: ID!): [Customers!]! # Retrieve Specific Customer by ID

# Orders

allOrders: [Orders!]! # Retrieve all Orders

specificOrders(ordersId: ID!): [Orders!]! # Retrieve Specific Order by ID

# Categories

allCategories: [Categories!]! # Retrieve all Categories

specificCategories(categoriesId: ID!): [Categories!]! # Retrieve Specific Category by ID

# Products

allProducts: [Products!]! # Retrieve all Products

specificProducts(productsId: ID!): [Products!]! # Retrieve Specific Product by ID

# Reorder

allReorders: [Reorder!]! # Retrieve all Reorders

specificReorders(reordersId: ID!): [Reorder!]! # Retrieve Specific Reorder by ID

# Inventory

allInventory: [Inventory!]! # Retrieve all Inventory

specificInventory(inventoryId: ID!): [Inventory!]! # Retrieve Specific Inventory by ID

# Orderlines

allOrderLines: [Orderlines!]! # Retrieve all Order Lines

specificOrderLines(orderlinesId: ID!): [Orderlines!]! # Retrieve Specific Order Line by ID

# Cust\_hist

allCustomerHistory: [Cust\_hist!]! # Retrieve all Customer History

specificCustomerHistory(cust\_histId: ID!): [Cust\_hist!]! # Retrieve Specific Customer History by ID

}

It’s in the name, calling these methods will retrieve the relevant information, for example, **allCustomers** will return every single Customer in the **Customers** table. The **specificCustomers** will return only the specified customer by providing the ID to the method.

Then, the **Mutation** type was specified in this file, this creates the methods that will manipulate the database.

###

# MUTATION

###

# Adding to Database

type Mutation {

# Create Customer

createCustomer(

firstname: String!,

lastname: String!,

address1: String!,

address2: String,

city: String!,

state: String!,

zip: String,

country: String!,

region: String,

email: String!,

phone: String,

creditcardtype: String,

creditcard: String,

creditcardexpiration: String,

username: String!,

password: String!,

age: Int!,

income: Float,

gender: String!

) : Customers

# Create Order

createOrder(

orderdate: DateTime!,

netamount: Float!,

tax: Float!,

totalamount: Float!,

customersId: ID!

) : Orders

# Create Category

createCategory(

categoryname: String!

) : Categories

# Create Product

createProduct(

title: String,

actor: String,

price: Float,

special: Boolean,

common\_prod\_id: Int,

# categoriesId: ID,

categoryname: String,

quan\_in\_stock: Int

) : Products

# Create Re-order

createReorders(

date\_low: DateTime!,

quan\_low: Int,

date\_reordered: DateTime!,

quan\_reordered: Int,

date\_expected: String,

productsId: ID!,

) : Reorder

# Create Inventory

createInventory(

quan\_in\_stock: Int,

sales: Int,

productsId: ID!

) : Inventory

# Create Order Line

createOrderline(

quantity: Int,

orderdate: DateTime!,

ordersId: ID!,

productsId: ID!

) : Orderlines

# Create Customer History

createCustomerHistory(

customersId: ID!,

ordersId: ID!,

productsId: ID

) : Cust\_hist

}

Lastly, the **types** were defined, these are straightforward re-definiations of the models specified in **datamodel.prisma**, except that the Prisma-specific directives have been removed.

###

# TYPES

###

# Customers Table

type Customers {

id: ID!

firstname: String!

lastname: String!

address1: String!

address2: String

city: String!

state: String!

zip: String

country: String!

region: String

email: String!

phone: Int

creditcardtype: String

creditcard: String

creditcardexpiration: String

username: String!

password: String!

age: Int!

income: Float

gender: String!

}

# Orders Table

type Orders {

id: ID!

orderdate: DateTime!

netamount: Float!

tax: Float!

totalamount: Float!

customers: Customers

}

# Categories Table

type Categories {

id: ID!

categoryname: String!

}

# Products Table

type Products {

id: ID!

title: String

actor: String

price: Float

special: Boolean

common\_prod\_id: Int

categories: Categories

inventory: Inventory

}

# Re-order Table

type Reorder {

id: ID!

date\_low: DateTime!

quan\_low: Int

date\_reordered: DateTime!

quan\_reordered: Int

date\_expected: String

products: Products

}

# Inventory Table

type Inventory {

id: ID!

quan\_in\_stock: Int

sales: Int

products: Products

}

# Order Lines Table

type Orderlines {

id: ID!

quantity: Int

orderdate: DateTime!

orders: Orders

products: Products

}

# Customer History Table

type Cust\_hist {

id: ID!

customers: Customers

orders: Orders

products: Products

}

**Part 2 – GraphQL Query Resolver**

*Build a GraphQL query resolver which returns some set of the attributes from a single database relation.*

For this part I had to return attributes from a table, I picked **Customers**, although I have all of the queries to return data from all of the tables for this lab.

const resolvers = {

/\*\*\*

\* Question (2)

\*

\* Build a GraphQL query resolver which returns some set

\* of the the attributes from a single database relation.

\*

\* Customers - No Relation

\* Categories - No Relation

\* Products - 1 Relation

\* Reorder - 1 Relation

\* Orders - 1 Relation

\* Inventory - 2 Relations

\* Orderlines - 2 Relations

\* Cust\_hist - 3 Relations - Question (3)

\*/

Query: {

/\*\*\*

\* CUSTOMERS

\*\*\*/

/\* Retreive all Customers \*/

allCustomers(root, args, context) {

return context.prisma.customerses()

},

/\* Retrieve a Customer with a Specific ID \*/

specificCustomers(root, args, context) {

return context.prisma.customerses({

where: {

id: args.customersId

}

})

},

As displayed in the code above, the **allCustomers** which is defined in the **schema.graphql** will return all of the customers and their attributes. It’s important to mention that the method called **customerses()** is a generated method, it looks at the table name which is **Customers** and tries to make it plural, if it cannot make it plural then it will add the “es” at the end of the table name.

In order to complete this question, I created sample data to enter into the **Customers** table with the following query which was entered in the **Playground**.

###

#

# Creating a Customer

#

###

mutation {

createCustomer(

firstname: "Gabriel"

lastname: "Grimberg"

address1: "Address 1"

address2: "Address 2"

city: "Some City"

state: "Some State"

zip: "zip1010"

country: "Ireland"

region: "Some Region"

email: "email2@email.com"

phone: "0833333333"

creditcardtype: "CC Type"

creditcard: "CC"

creditcardexpiration:"08/19"

username: "eadlab3"

password: "e3kj4rewdnergjkn"

age: 21

income: 100000.00

gender: "Male"

) {

id

firstname

lastname

address1

address2

city

state

zip

country

region

email

phone

creditcardtype

creditcard

creditcardexpiration

username

password

age

income

gender

}

}

Then, I queried the database to display back to me all of the **Customers**.

###

#

# View All Customers

#

###

query {

allCustomers {

id

firstname

lastname

username

email

}

}

I also created a query as mentioned to display a specific **Customer**, all I have to do is supply the ID.

###

#

# View a Specific Customer by ID

#

###

query {

specificCustomers(customersId: "cjtexggou00m00771s954gjbb") {

id

firstname

lastname

address1

address2

city

state

zip

country

region

email

phone

creditcardtype

creditcard

creditcardexpiration

username

password

age

income

gender

}

}

It’s important to note that you do not have to specify all of the fields to be returned, you can query to return for example just the ID and the username.

**Part 3 – Three Joined Database Relations**

*Build a GraphQL query resolver which returns the attributes from 3 joined database relations having 2 levels of nesting in the resultant output*

*Briefly, describe an application of the query you have chosen to write as a comment in your resolver code*

For this part, I had to use a table which had three relations the only table which I picked was the **Cust\_hist** table as it had three relations to it. Customers, Orders and Products.

Similar to the question above, I had to return the results.

/\*\*\*

\* CUST\_HIST

\*\*\*/

/\*\*\*

\* Question (3)

\*

\* Build a GraphQL query resolver which returns the attributes from 3 joined database relations

\* having 2 levels of nesting in the resultant output.

\*

\* Joined Tables:

\* - Customers

\* - Orders

\* - Products

\*

\* Description:

\* - A method to display all the customer's history and a method

\* to display a specific customer history with the given ID.

\*

\* - Customer History is like a receipt, it will have it's own unique ID

\* and it will include the customer, order and product details.

\*

\* - This has 3 joined database relations and 2 levels of nesting in the output.

\*/

/\* Retrieve all Customer History \*/

allCustomerHistory(root, args, context) {

return context.prisma.cust\_hists()

},

/\* Retrieve a Customer History with a Specific ID \*/

specificCustomerHistory(root, args, context) {

return context.prisma.cust\_hists({

where: {

id: args.cust\_histId

}

})

}

}, // End Query

To make the connections between these tables I did the following:

/\*\*\*

\* Question (4)

\*

\* This links the following tables: Customers, Orders and Products

\* with the Cust\_hist table.

\*/

/\* The link for Customer History, Customers, Orders and Products \*/

Cust\_hist: {

customers(root, args, context) {

return context.prisma.cust\_hist({

id: root.id

}).customers()

},

orders(root, args, context) {

return context.prisma.cust\_hist({

id: root.id

}).orders()

},

products(root, args, context) {

return context.prisma.cust\_hist({

id: root.id

}).products()

}

}

} // End Resolvers

**Part 4 – Mutation Resolver**

*Create a mutation resolver to add data the database. Your mutation should update at least two relations (of your choice). Briefly, describe an application of the query you have chosen to write as a comment in your resolver code*

I used the **createProduct(…)** as an example to solve this question.

Upon creating a new **Product** a new **Inventory** and **Category** (if applicable) will be creating alongside this solves the problem of updating at least two relations.

/\*\*

\* Question (4)

\*

\* Create a mutation resolver to add data the database.

\* Your mutation should update at least two relations (of your choice)

\*

\* Description:

\* - Upon creating a new Product, a new Inventory will be created for it.

\* - Alongside this, Category can also be created and linked with this Product.

\*

\* Relations Updated:

\* - Categories

\* - Inventory

\*/

/\* Creating a Product \*/

createProduct(root, args, context) {

return context.prisma.createProducts({

title: args.title,

actor: args.actor,

price: args.price,

special: args.special,

common\_prod\_id: args.common\_prod\_id,

categories: {

// connect: { id: args.categoriesId },

create: { categoryname: args.categoryname }

},

inventory: {

create: { quan\_in\_stock: args.quan\_in\_stock, sales: 0 }

}

},)

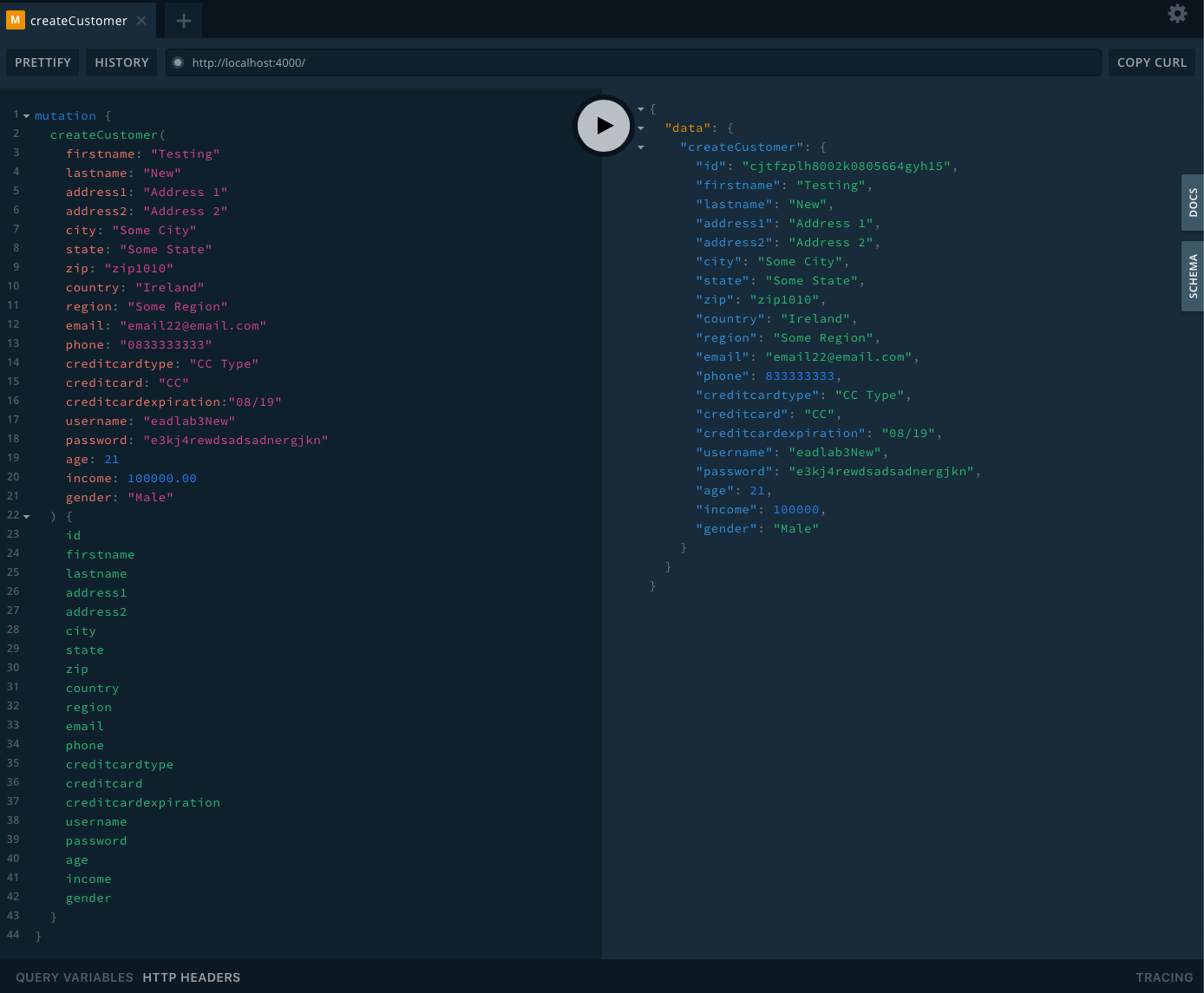
},

**Part 5 – GraphQL Server and Test**

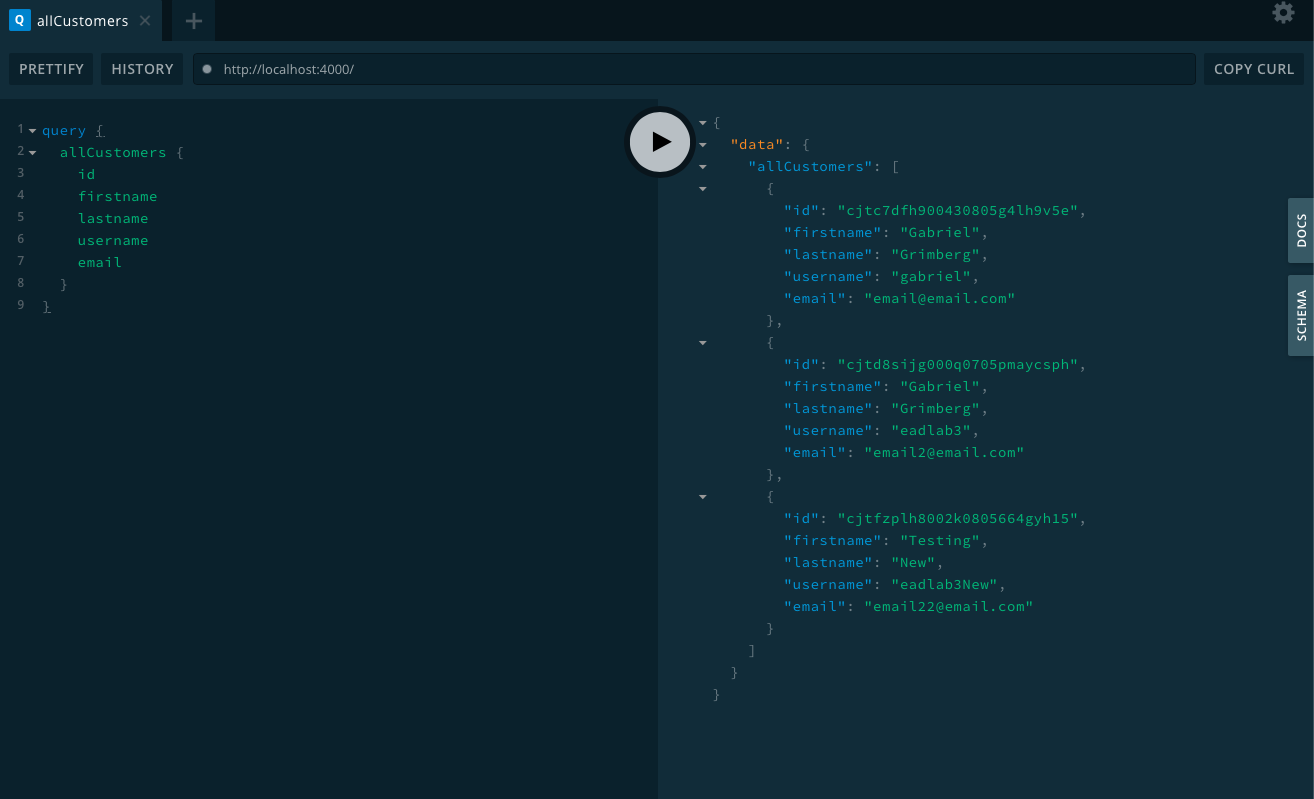
Set up a running GraphQLServer from the graphql-yoga library to test and demonstrate your resolver queries and mutations you implemented in sections 2-4 above

This last part involves setting up the server successfully where you have a **Playground** to test your queries and mutations.

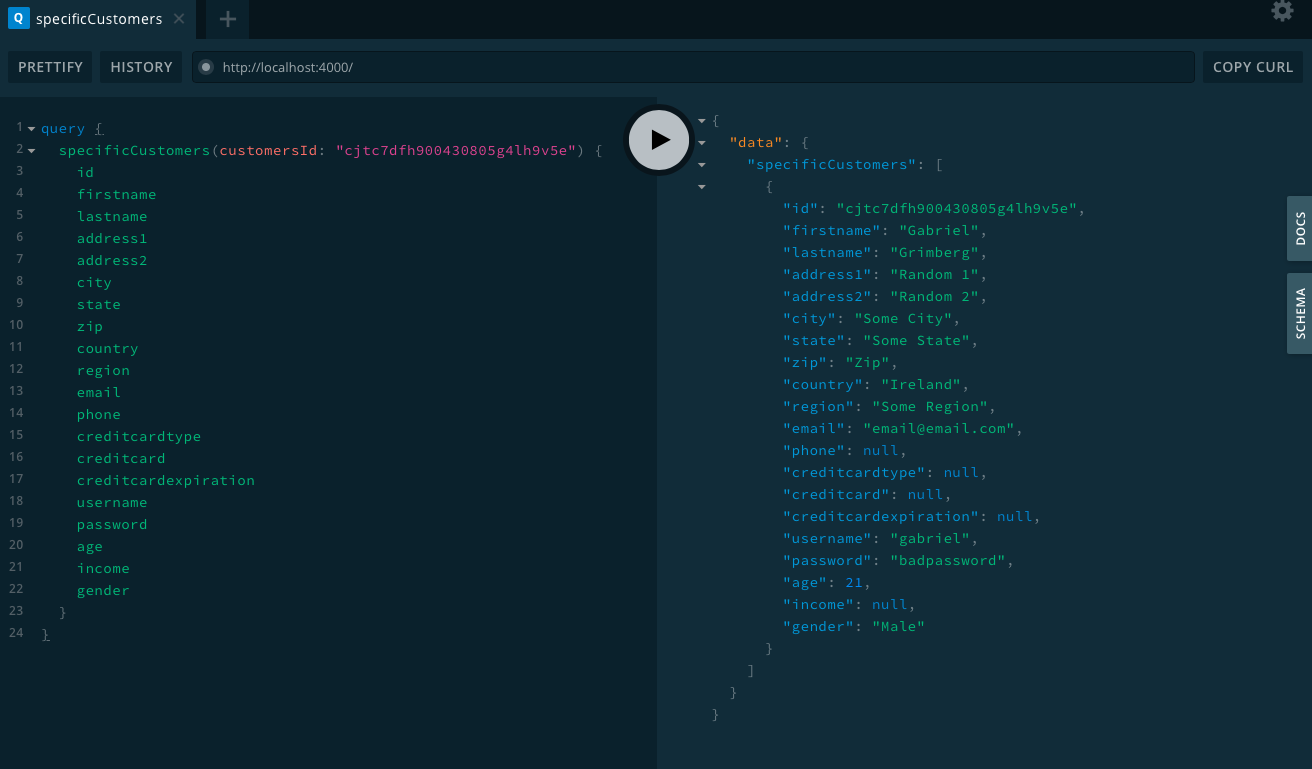
**Part 2 – Creating Customer**

****

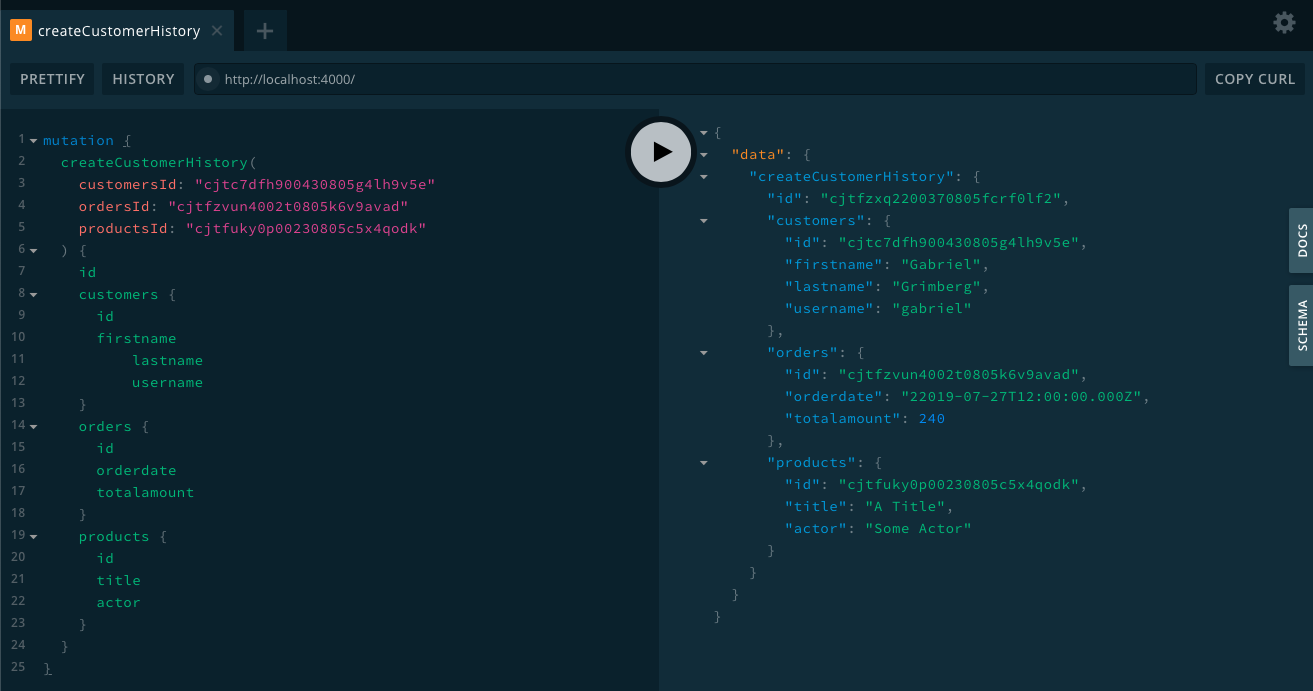
**Part 2 – View All Customers**



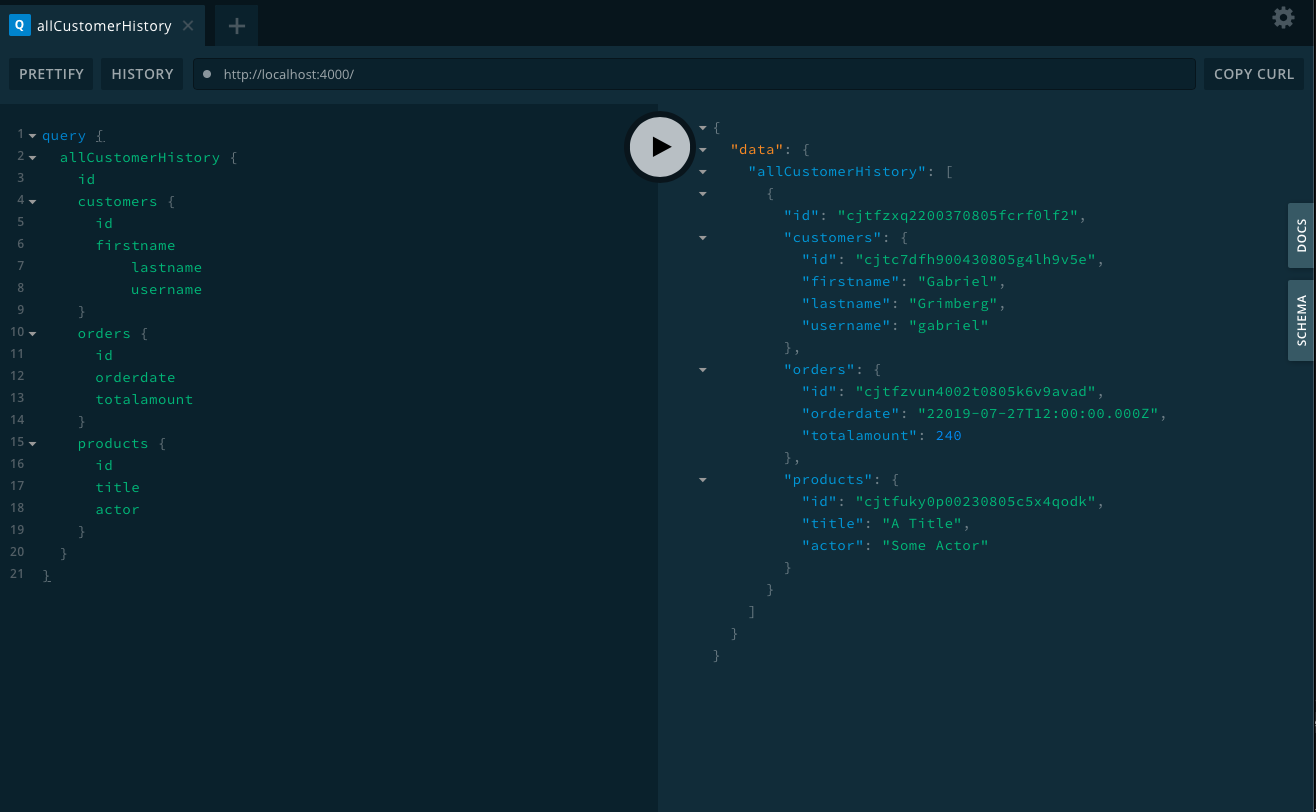
**Part 2 – View Specific Customer**



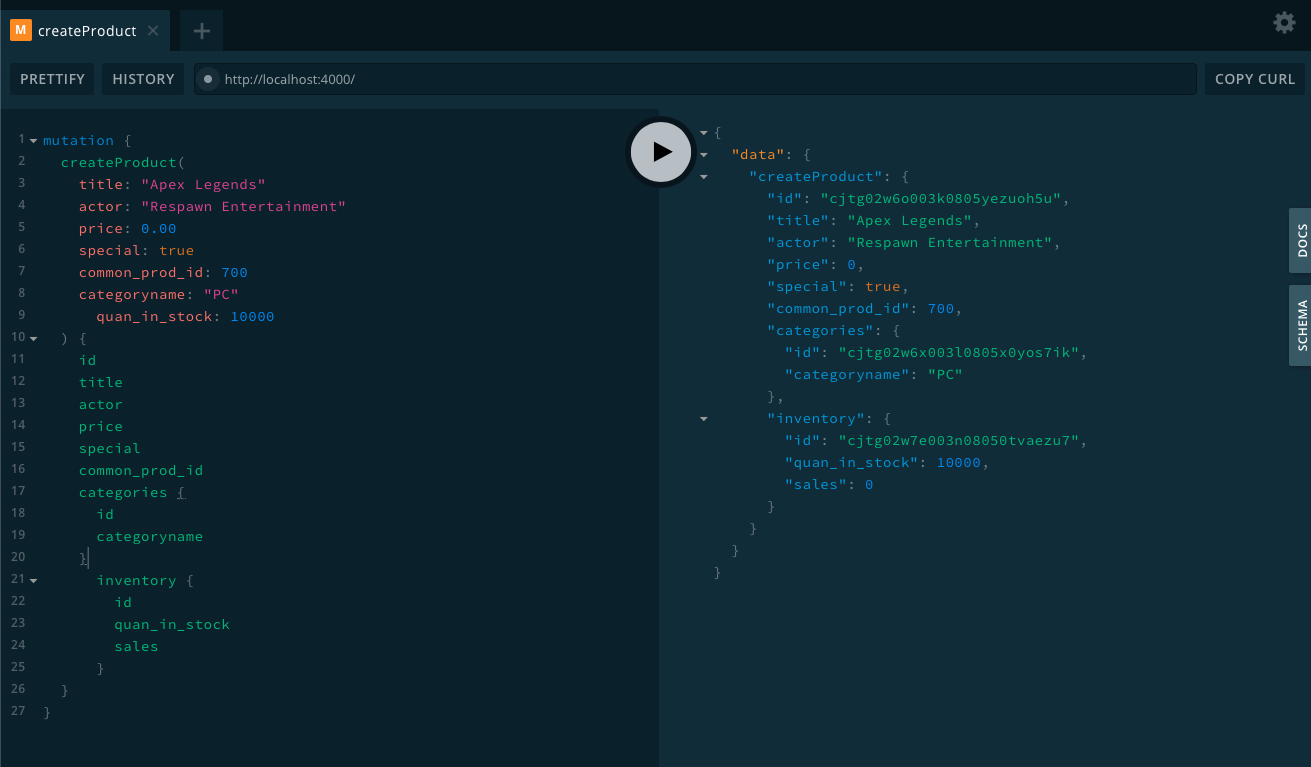
**Part 3 – Create Customer History (3 Relations + 2 Levels Nesting)**

****

**Part 3 – View Customer History (3 Relations + 2 Levels Nesting)**

****

**Part 4 – Create Product (Update 2 Relations)**

****